On Introducing Built-In Test for Software Components in AADL Models

Valérie-Anne Nicolas
Université Européenne de Bretagne
Université de Brest
Laboratoire d'Informatique des SYstèmes Complexes (LISYC)
20 avenue Le Gorgeu - CS 93837 - 29238 Brest Cedex 3 – France
vnicolas@univ-brest.fr

Abstract

This paper presents preliminary ideas to include a kind of built-in self-test for systems embedded software components. The study promotes contract-based testing applied to AADL modeling of hardware/software systems. The aim is first to help evaluating the testability of software components embedded in such systems, and next to improve the integration step, especially in the context of COTS design and development. We introduce an architecture to include a generic test system inside an AADL model, and then test specifications to handle the testing process. The paper exposes the main ideas of the proposed approach and its modeling but no implementation work. Next stage is an implementation work to assess the feasibility and the benefits of the proposed approach.

1. Introduction

COTS (Commercial Off-The-Shelf) based system design and development is a way to build new systems using already available components. Since the 90's, this technology is more and more widely used as it reduces components cost and development time, and promises better system performance and quality. However, experience revealed some drawbacks: increase in software component integration work, lack of precise specification and control of the components. The difficulty to figure out what is the real behavior of a component in a specific context leads to a complex yet not safe integration work.

We believe that testing is a way to tackle these difficulties. This paper presents a built-in self-test way to apply contract-based software testing [7] to embedded software components in systems of hardware and software components. We use AADL [1] for system modeling purpose.

The paper is organised as follows: we first introduce the AADL language in section 2, and next contract-based testing in section 3. In section 4, we sketch our approach. In section 5, we present the proposed test architecture and the associated test engine and test specifications. We end with a discussion and a conclusion.

2. AADL Language

The AADL language (Architecture Analysis and Design Language) is a component-based hierarchical architecture design language [1]. AADL is a SAE standard [2] whose first version V1.0 was published in 2004, and the second version V2.0 in 2009. The language benefits from an active community and is still evolving, being improved and including new functionalities and tools supported by the Open Source AADL Tool Environment (OSATE [3]).

First intended for avionics specific needs, AADL offers facilities to specify and model real-time embedded systems following an MDE (Model-Driven Engineering) approach. It allows the description of hardware and software components of distributed real-time embedded systems in a graphical and textual framework. A component (system, process, thread, subprogram, data, processor, bus…) is specified by an interface and properties (e.g. behavior, source code…). Components are linked by connections.

Applications in the scope of AADL have a high level of criticality and have to deal with hard constraints while executing (tasks scheduling…). AADL allows the specification of behavioral properties of components for specific analyses (e.g. non-functional properties such as temporal constraints) or

automatic validation purpose using language annexes. Execution platforms also exist, based on code generators for several languages and associated runtimes [4]. From an AADL model it is then possible to automatically generate an executable code. Running it allows to collect or verify some functional properties about the system integration. One may think that it could also be used to run test data. But to our knowledge, testing needs are addressed by neither AADL core language nor annexes.

3. Contract-based testing

In component-based software engineering (using COTS), choosing the right component to achieve a given role in a specific integration context is not so easy. User must know, from the design phase, if a potential component fits his integration constraints. In other words, he needs a specification stating very precisely the component's behavior and using context, without entering implementation details. This is the notion of contract of a component, introduced in the Design by Contract approach for oriented-object languages [5]. This approach is especially implemented by the Fractal Architecture Description Language [6]. Beside this, contracts are also used to build self-testable classes for object-oriented languages [7]. We believe that this work can be transferred to AADL software components.

4. Built-In Self-Test for AADL

4.1. Context

As mentioned in section 2, AADL is dedicated to highly critical system design. AADL tools focus on the analysis of critical non-functional properties resulting from the integration of components, leaving aside the behavior of single components. This methodological choice is entirely consistent with COTS based system design and development, where components should be considered safe and reliable. However, experience shows that this assertion is not always guaranteed because of specification incompletness concerning system integration. Even if a single component may be considered safe, what about its real behavior in a specific integration context? What are the true context requirements (preconditions) needed by a component? These are real problems encountered during system integration of COTS components. They are especially true in AADL where bindings between software and hardware components are specified during the design stage. To face these difficulties, we believe it should be useful to be able to evaluate a software component behavior, on well chosen data, embedded inside its specific software/hardware integration context. The test approach sketched next section goes in that direction, and even further as it may also be used for maintenance.

4.2. Proposed test approach

The study outlined here is about a built-in self-test approach for AADL models. The idea is to adapt the hardware BIST concept to AADL software architectures. We chose the built-in characteristic to fit with AADL purposes of integration, reuse and validation during the design stage. Another reason is that it greatly facilitates future maintenance of the system. The self-test aspect is intended to automate as much as possible the test effort. Nowadays, most hardware components include BIST. That is why we focus here on the software parts of an AADL model. Unlike hardware, there is no generic software fault model. We therefore propose to use contracts for the user to specify testing information such as test criterion, test coverage type and rate, to enable embedded software components self-testing process. As stated in section 3, contract-based software testing is an instance of the BIST concept for software object-oriented languages. One goal of this study is to assess the feasibility of this kind of approach for software components in AADL models.

Next section, we first introduce an architecture to include BIST software facilities in AADL models, then a test engine to achieve the testing stages and test specifications (so-called test contracts) to describe testing process properties.

5. An architecture for AADL Built-In Self-Test

5.1. Test architecture

We define a BIST AADL system (Figure 1) as an architecture incorporating the AADL model of the application to be tested (so called user application) and a generic test model. The generic test model is the test engine providing the needed testing functionalities. To achieve a testing stage, the test engine must use the model of the user application. The execution of a BIST AADL system is parameterized by two operation modes (interface features): standard mode and test mode. Standard mode consists in executing the user application alone, the test engine being disabled. Test mode is to run the test engine, which itself executes the

user application as needed, in order to manage the testing process.

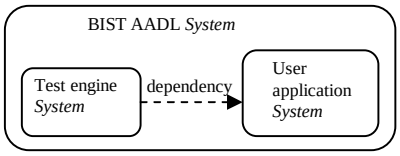

Figure 1: BIST AADL system architecture

5.2. Test engine

Test engine has to manage the overall testing process of software parts of an AADL model. It should offer facilities to generate test data, to run user application on these test data, to collect and interpret results (oracle), and finally to produce a test report.

We modeled the test engine as an AADL system with four processes: the monitor, the generator, the runner and the oracle (Figure 2). Each process contains a thread to achieve its task using subprograms. The test engine system also has a processor component and a bus component. We use AADL properties to connect the elements of the interfaces of the different processes, to link the processes and the bus to the processor and to connect with the bus. Using the AADL component data, we define four data types to model the different kind of information exchanged in the system: *aadl_file*, *test_data*, *output_data*, *test_report.*

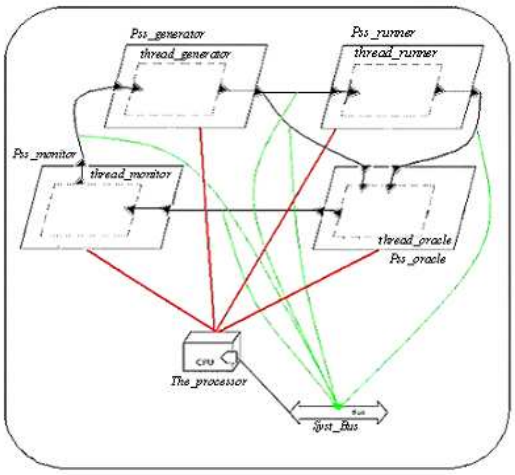

Figure 2: Test engine system model

The monitor launches the testing process by sending to the generator the AADL user application model (*aadl_file*) to be tested, after that it will await the associated test report (*test_report*) from the oracle.

The generator analyses the input AADL model (*aadl_file*) in order to retrieve test specifications and to produce test data and associated expected results (we give more details on this stage in section 5.3, after introducing test specifications). Test data (*test_data*) are then sent to the runner and expected results (*output_data*) to the oracle.

The runner executes the user application on all the different test data (*test_data*) and saves the obtained results. It should be noted that this does not just unit testing but rather context embedded testing. These results (*output_data*) are sent to the oracle at the end of the running stage.

The oracle compares its two inputs (*output_data*), expected results from the generator and obtained results from the runner, to produce a test report (*test_report*) and send it to the monitor.

5.3. Test specifications

In order to have a relevant testing process, AADL models must contain information to guide the test engine. More precisely, the generator needs some test specifications to produce test data (cf. section 5.2). As we focus on AADL software parts testing, the test specifications have to be located in subprogram components. These test specifications should permit to define one's component testing policy: test strategies, criteria, metrics, expected results…

We propose a preliminary library (*TestPropertySet*) providing new AADL properties for specifying testing features of subprogram components. We distinguish four types of properties: *test_strategy*, *test_criterion*, *test_coverage*, *test_data_file*.

Test_strategy property may be *DT_predefine* or *DT_criterion_applying*. *DT_predefine* means that a file containing test data with associated expected results already exists and is available at the location written in the property *test_data_file*. *DT_criterion_applying* means that the test engine has to apply a given criterion, specified in property *test_criterion*, on the source code of the subprogram to generate test data.

Test_criterion property enables the choice of a structural test criterion if *test_strategy = DT_criterion_applying* (e.g. *instruction_testing*, *branch_testing*, *path_testing*, *mutation_testing*…).

Test_coverage property states the criterion coverage rate needed for the test data generation step. For example a value below 100% allows dead code in the subprogram. It could also be useful when the coverage of a criterion can not be determined statically without approximation. This property is enabled when *test_strategy = DT_criterion_applying*.

Test_data_file property specifies the location of the file needed when *test_strategy = DT_predefine*.

We focus on safety testing, specific real-time distributed embedded properties being studied by dedicated AADL tools. As the source code of the subprogram is available in the AADL model, we use classical software structural testing criteria. Functional criteria could also be used if a behavioral description of the subprogram is available.

From considered test criteria, automatic test data generation is easily done, but associated expected test results production remains a problem. To face this, the user may be asked to provide expected results to the test engine. This solution presents a lot of disadvantages: not automatic, time expansive, error prone, and difficult especially concerning embedded third-party components. Otherwise an executable specification or a behavioral model of the component is needed to automatically compute the expected test results, using model-based testing methods [8].

These component embedded test specifications are needed to produce test data, but they also may be seen as contracts, allowing knowing by what criterion and to what extent a component is well tested. They may also be used to assess component's testability. In fact, we consider that test specifications are a tool to make testable components.

6. Discussion

We chose to add a BIST part in an AADL model at the system level in order to be able to integrate test data but also the test engine for every user application with a minimum extra cost for both the user and the system. In fact, test engine is generic, and if an implementation of the test engine is available as an AADL system, it is no cost for the user to package it with his model to obtain a BIST version. The only additional effort remaining for the user would be to add test specifications in his model. Another important advantage is that the application could be tested in-line in any new environment of deployment for design purpose but also during maintenance. For systems with very strong memory constraints, the extra memory cost of the test engine could already be too important. In that case, a BIST version of the system could not be deployed on a real target, however it could be used during the design stage and off-line maintenance.

A different option would be to add the BIST part at the component level but it would lead to a lot of redundancy as the test engine may be the same for the different components and to a memory overcost unacceptable for such real-time embedded systems. Moreover, some testing functionalities need to be at the system level in order to achieve integration testing.

7. Conclusion

This study shows that introducing test specifications (or test contracts) in software parts of AADL models is a way to define their testability and could ease the integration step. Combined with a test engine at the system level, it could generate and run test data automatically in the manner of a BIST during design and maintenance stages. One advantage is that software parts are tested in their integration context.

However, to broaden the range of supported test criteria, to enable the oracle and to fully automate the testing stage, behavioral models of subprograms are required. The previously proposed library for test specifications is to illustrate the approach and also needs to be enriched (e.g. properties to declare the hardware/software binding constraints). Next stage is an implementation to assess the feasibility and the benefits of the proposed approach.

8. References

[1] P.H. Feiler, D.P. Gluch, and J.J. Hudak, "The Architecture Analysis & Design Language (AADL): An Introduction", *Technical Note CMU/SEI-2006-TN-011*, February 2006.

[2] SAE, "Architecture Analysis & Design Language (AADL)", *SAE Standard AS5506A*, January 2009.

[3] The SEI (Software Engineering Institute) AADL Team, "An Extensible Open source AADL Tool Environment (OSATE)", *Technical Note SEI*, June 2006.

[4] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite", *ACM Transactions in Embedded Computing Systems (TECS)*, October 2008.

[5] B. Meyer, *Object-Oriented Software Construction, 2nd Edition*, Prentice Hall, New Jersey, 1997.

[6] "Component-based architecture: the Fractal initiative", *Annals of telecommunications*, Institut Telecom, Springer, vol. 64, no 1-2, January-February 2009.

[7] D. Deveaux, J-M. Jézéquel and Y. Le Traon, "Reliable Objects: Lightweight Testing for OO Languages", *IEEE Software*, July 2001.

[8] M. Utting, and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann Publishers Inc., San Francisco, 2006.